УДК 681.3.06

С.Ю. ГАВРИЛЕНКО**,** канд. техн. наук, НТУ "ХПИ",
А.М. ФИЛОНЕНКО**,** канд. техн. наук, НТУ "ХПИ",
Р.Ю. КОМАРЕВ**,** НТУ "ХПИ"

ОПТИМИЗАЦИЯ ОРГАНИЗАЦИИ ТАБЛИЦ СИМВОЛОВ ПРИ ПОСТРОЕНИИ КОМПИЛЯТОРОВ

Розглянуто основні типи організації таблиць символів при побудові компіляторів. Наведено характеристику видів таблиць символів та методів оптимізації по швидкодії та за обсягом. Запропонована модель, яка використовує підпорядковані таблиці, що дозволяє відокремити глобальні та локальні ідентифікатори і таким чином зменьшити час доступу до ідентифікатора та також зменьшити ймовірність колізій.

The basic types of the organization of tables of symbols are considered at construction of compilers. The characteristic of kinds of tables of symbols and methods of optimization on speed and on volume is resulted. The model which uses the subordinated tables is offered, resolving to separate global and local identifiers and thus to reduce time of access to the identifier and also to reduce probability of collisions.

Постановка проблемы. Несмотря на широкое разнообразие языков программирования и интегрированных сред программирования, оптимизация работы программ-компиляторов остается актуальной темой дискуссий для многих разработчиков программного обеспечения.

Проверка правильности семантики и генерация кода требуют знания характеристик идентификаторов, используемых в программе на исходном языке. Эти характеристики выясняются из описаний и из того, как идентификаторы используются в программе, и накапливаются в таблице символов или в списке идентификаторов. Эффективность взаимодействия компилятора с таблицей символов напрямую зависит от способа организации, построения и метода поиска данных. То есть перед нами стоит конкретная проблема – проблема организации таблиц символов.

Анализ литературы. Таблицы всех типов имеют общий вид, приведенный на рис. 1, где слева перечисляются аргументы, а справа – соответствующие значения [1, 2, 3].

Рис. 1. Общий вид таблиц символов

Каждый элемент обычно занимает в машине более одного слова. Если элемент занимает *k* слов и нужно хранить ***N*** элементов, то необходимо иметь *k*N* слов памяти. Расположить информацию можно двумя способами:

1. Каждый элемент в *k* последовательных слов и иметь таблицу из *k*N* слов.

2. Иметь *k* таблиц, например, *T*1*, T*2*, ..., Tk*, каждая из ***N*** слов. Весь *i*-й элемент при этом будет находиться в словах $T1_i$, ..., Tk_i.

В нашем частном случае аргументами таблицы являются символы или идентификаторы, а значениями – их характеристики. Так как число литер в идентификаторе непостоянно, в аргументе часто помещают вместо самого идентификатора указатель на идентификатор. Это сохраняет фиксированный размер аргумента.

Идентификаторы хранятся в специальном списке строк. Число литер в каждом идентификаторе может храниться как часть аргумента или в списке идентификаторов прямо перед идентификатором. Оба способа, на примере таблицы, содержащей элементы для идентификаторов *I*, *MAX*, и *J*, показаны на рис. 2

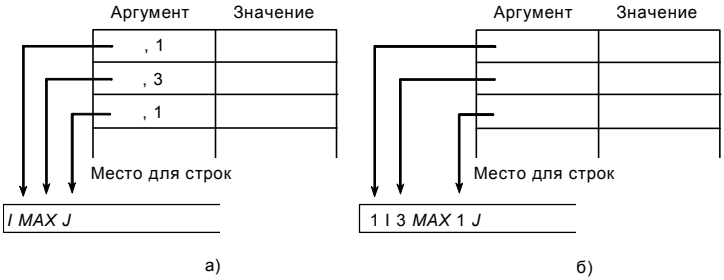

Рис. 2. Способы запоминания аргументов

Так как при компиляции на процесс поиска необходимых данных уходит много времени, важно выбрать такую организацию таблиц, которая допускала бы эффективный поиск.

Неупорядоченные и упорядоченные таблицы. Простейший способ организации таблицы состоит в том, чтобы добавлять элементы для аргументов в порядке их поступления, без каких-либо попыток упорядочения [3 – 6]. Поиск в таком случае требует сравнения с каждым элементом таблицы, пока не будет найден подходящий.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку аргументов. В нашем случае, где аргументами являются строки литер, наиболее естественным является упорядочение, порождаемое внутренним представлением строк литер. Обычно оно совпадает с лексикографическим порядком. Так, строки *A, AB, ABC, AC, BB* расположены в возрастающем

порядке. Эффективным методом поиска в упорядоченном списке из ***n*** элементов является так называемый бинарный или логарифмический поиск.

Хеш-адресация – это метод преобразования символа в индекс элемента в таблице (элементам присваиваются номера 0, 1, 2, …, $N - 1$, если таблица состоит из N элементов) [7]. Простой хеш-функцией является внутреннее представление первой литеры символа. Так, если двоичное представление *A* есть 11000001, то результатом хеширования символа *ABE* будет код 11000001 (*C*1 в шестнадцатеричной системе счисления). Начальным индексом, с которого начинается поиск элемента для аргумента *ABE*, будет 11000001.

Пример хеш-адресации для идентификаторов *ABE*, *B* и *I* показано на рис. 3.

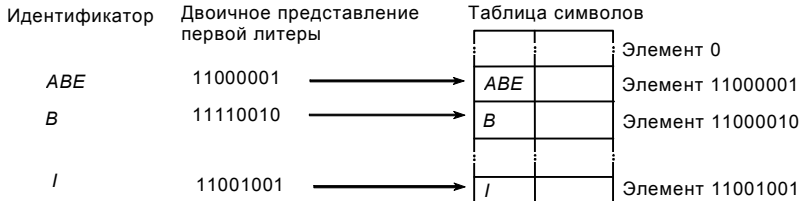

Рис. 3. Хеш-адресация

Таблицы символов, имеющие структуру дерева. В этом методе для упорядочения элементов используется бинарное дерево, в котором к каждому узлу может быть "подвешено" не более двух поддеревьев. Каждый узел дерева представляет собой заполненный элемент таблицы, причем корневой узел является первым элементом [6].

Таблицы символов, имеющие блочную структуру. Некоторые языки имеют структуру вложенных блоков и процедур [8]. Один и тот же идентификатор может быть описан и использован много раз в различных блоках и процедурах, и каждое такое описание должно иметь единственный, связанный с ним элемент в таблице символов. При использовании идентификатора возникает проблема, как найти соответствующий ему элемент в таблице символов.

Правило нахождения соответствующего идентификатору описания состоит в том, чтобы сначала просмотреть текущий блок (в котором идентификатор используется), затем объемлющий блок и т. д. до тех пор, пока не будет найдено описание данного идентификатора. Мы можем осуществить такой поиск, храня все элементы таблицы для каждого блока в смежных ячейках и используя список блоков.

Цель статьи. Оптимизация компилятора за счет использования подчиненных таблиц, что позволяет отделить глобальные и локальные

идентификаторы и таким образом уменьшить время доступа к идентификаторам и количество коллизий.

Проанализировав основные виды организации таблиц символов при построении компиляторов, можно предложить новый метод организации таблиц символов с использованием подчиненных таблиц. Основная мысль предлагаемой конструкции и методов работы с ней заключается в том, чтобы объединить положительные качества всех, рассмотренных ранее, стандартных методов.

На рис. 4 изображено использование одной основной таблицы для записи глобальных идентификаторов, и подчиненных таблиц для локальных идентификаторов. При этом доступ к данным основной и подчиненных таблиц осуществляется посредством хеш-адресации. Таким образом, мы объединяем метод хеш-адресации, структуру типа дерево и блочную структуру.

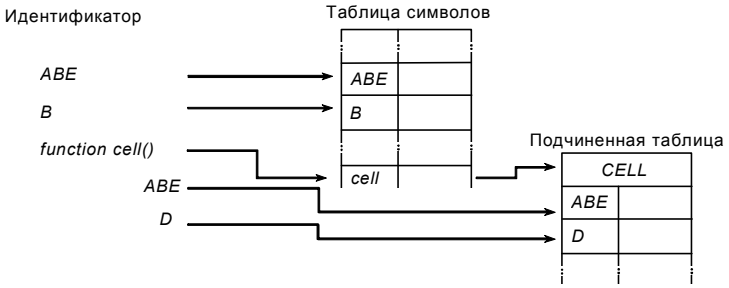

Рис. 4. Использование подчиненных таблиц

Сравним различные методы работы с таблицами по времени поиска. Мы будем проводить сравнение в терминах ожидаемого числа E сравнений аргументов, которые нужно выполнить, чтобы найти данный символ. Это число, зависящее от коэффициента загрузки *lf (load factor)* таблицы в данный момент, представляет собой отношение текущего числа элементов n к максимально возможному числу элементов N (1)

$$lf = n / N. \quad (1)$$

Благодаря использованию хеш-адресации в пределах таблицы время поиска практически совпадает со временем, затраченным на хеширование. Так мы получаем огромную экономию по сравнению со временем поиска по неупорядоченной таблице из n позиций,

$$E = n / 2 \quad (2)$$

или даже по сравнению с временем поиска по упорядоченной таблице

$$E\max = 1 + log_2 n. \quad (3)$$

Одновременно с этим, создавая подчиненные таблицы для записи локальных переменных, мы значительно уменьшаем вероятность коллизий, возникающих в результате использования программистом одинаковых идентификаторов в основной программе и подпрограммах. Также, за счет малой глубины вложенности подчиненных таблиц, удалось убрать лишние расходы времени на переход от одного узла к другому, что неизбежно происходит при организации таблицы по типу "дерево". И, наконец, нам удалось избежать избыточного описания нахождения идентификатора в таблице, как было при использовании блочной структуры.

Предложенный алгоритм с учетом математических выкладок апробировался на четырех видах задач:

1) программный код объемом в 2 "страницы" с несколькими простыми функциями;

2) программный код объемом в 10 страниц и большим числом простых функций;

3) программный код объемом в 10 страниц с использованием сложных объектов и активным взаимодействием между функциями;

4) программный код объемом в 50 страниц с использованием сложных объектов, большим числом функций и их активным взаимодействием между собой.

Все виды программных кодов последовательно запускали на специально разработанных для этих целей компиляторах. В итоге получили достоверную статистику по скорости обработки данных разного типа и объема компиляторами с различной организацией таблиц символов.

Таблица

Скорость обработки данных

Скорость обработки апробированных задач			
Задача № 1	Задача № 2	Задача № 3	Задача № 4
Одноуровневая таблица с использованием хеш-адресации			
0,2 секунды	1,2 секунды	2,5 секунды	8 секунд
Организация типа дерево			
0,2 секунды	2,2 секунды	4,5 секунды	19 секунд
Блочная структура			
0,7 секунды	4,2 секунды	5,5 секунды	25 секунд
Организация с использованием подчиненных таблиц			
0,3 секунды	1,7 секунды	2,5 секунды	6 секунд

Выводы. Данные, полученные в результате апробации, свидетельствуют о целесообразности использования предлагаемого метода для идентификации сложных объектов, имеющих объемный исходный код. Предложенный метод может быть применен в научных исследованиях, в коммерческих целях, а также широким кругом пользователей, занимающихся увеличением скорости обработки данных разного типа и объема.

Список литературы: 1. *Ульман Джеффри, Ахо Альфред, Сети Рави.* Компиляторы: принципы, технологии и инструменты. – М.: Издательский дом "Вильямс", 2001. – 768 с. **2.** *Кнут Д.* Искусство программирования. Т. 1. Основные алгоритмы. – М.: Издательский дом "Вильямс", 2005. – 823 с. **3.** *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1975. – 327 с. **4.** *Карпов В.Э.* Классическая теория компиляторов. – М.: МГИЭТ, 2003. – 61 с. **5.** *Рэй Дункан.* Оптимизация программ на ассемблере. PC Magazine/Russian Edition. – 1992. – .№ 1. – P. 102–117. **6.** *Хантер Р.* Основные концепции компиляторов. – М.: Издательский дом "Вильямс", 2002. – 256 с. **7**. *Вайнгартен Ф.* Трансляция языков программирования. – М.: Мир, 1977. – 190 с. **8.** *Левитин В., Ананий В.* Алгоритмы: введение в разработку и анализ. – М.: Издательский дом "Вильямс", 2006. – 576 с.

Поступила в редакцию 20.04.2007